SESPS-SABRE Event Builder

Gordon McCann
Contributions from: Ken Hanselman, Erin Good,
Sudarsan Balakrishnan, Kevin Macon, Chris Esparza

July 2021

1 Introduction

This paper is aimed at giving a description of the functionality of the SESPS-SABRE Event Builder and a brief example of how to use the EventBuilder code. Note that this code is a *template*. I will outline a general method by which the event builder will work, but most experiments will require the user to tweak the code to fit their individual data set. Additionally I will try to give pointers on how the user can modify the code to better fit other use cases. Hopefully this will allow everyone to work on SPS-SABRE data from the same basic template!

Some notes about compatibility: The current version has been tested on Ubuntu 20.04, SL 7, and Mac OSX Catalina using ROOT 6. Any other combination of software is the wild west.

2 Event Building

First, some jargon: a hit refers to a single signal in a single channel of a detector setup, while an event is a collection of coincident hits. Event building is the concept of creating time-ordered data by grouping together coincident detector hits into events. Thus the goal of the event builder code is to take data from the CoMPASS DAQ and generate these ordered events for use in further physics analysis.

CoMPASS gives raw binary data stored in a file for every channel in the system (read: if there are a 147 channels, there are 147 files). While each of these individual files is ordered in time, there is no information about how the files are ordered relative to each other. The main hurdle of the event builder is then to read in the data from all of these files simultaneously, and then perform an insertion sort-like operation upon the data to generate a single file of ordered and organized data.

The event builder as it currently exists is not *strictly* an event builder. It has a final analysis stage (ConvertSlowA or ConvertFastA) which generates a large amount of physics data. This can then be fed into the plotting tool to generate a large number of diagnostic plots. The reason that this end of the pipeline exists is primarily for analysis while actively running an experiment. It is recommended that the user build a separate analysis code for the data after the event building is completed.

CoMPASS raw data should be given names of the format `run_#` and be tarred up into `.tar.gz` format and stored in the `raw_binary` directory of the workspace. Event-built data is currently stored using ROOT, for both ease of further analysis as well as efficient storage. For convenience, a shell script called `archivist` is in the bin directory. The archivist can tar up files from the CoMPASS data directory and transfer them to a event builder workspace. To use archivist simply modify the paths to the CoMPASS directory and the workspace of the event builder.

The event builder requires a workspace with a predefined set of directories. The workspace directory should contain the following sub-directories: `raw_binary temp_binary sorted fast analyzed histograms`

3 Code Details

3.1 CompassFile and CompassRun

CompassFile and CompassRun are the two main classes which interface with the compass binary data. The data from the tar archive is unpacked to the `temp_binary` directory, and then a list of CompassFiles is opened by the CompassRun, where each file contains a fixed sized buffer; the size of the buffer is currently defined in `CompassFile.h` using the `bufsize` variable. The size is defined as number of hits to be read, where a hit in the SPS-SABRE setup is 24 bytes long. The larger the buffer, the more efficient the program is, to a certain limit. If the buffer is larger than the size of the largest file, there are no additional gains to speed; if the buffer is too large, fast memory (read: RAM) will be exceeded and slower memory will have to be used. In this way, it is typically best to use a several megabyte size, to account for the larger files, while not exceeding most systems memory limits. The default value is `bufsize=200000`, which corresponds to 4.8 MB per file. Methods exist for allowing for a user determined bufsize at runtime, but they are currently not implemented in the SPS-SABRE method, as this is not frequently needed to change.

3.2 Sorting

There are two sorting methods: Slow and Fast. The slow method is simply a single coincidence window which grabs together all hits within a window and creates an event. The window is started by the first event; that is no one specific channel is the trigger. Channels can have their timestamps shifted by a specific value using the ShiftMap feature, which allows the user to move hits to reduce deadtime. Typically, SPS-SABRE requires a shift of the scintillator and the SABRE channels to match with the anode time. This reduces deadtime, and leads to a more efficient pipeline.

Raw channel data is associated with specific detector variables using two stages. The first is the channel map, which converts the global channel number (boardNumber*16 + channel) with a key to a variable map held by the SlowSort class. This means that to add a new type of detector, the ChannelMap class and the SlowSort class would need modification (code contains comments on how to modify). This is somewhat cumbersome, however it is quite efficient from a optimization perspective.

After slow sorting, the code can optionally run the fast sorting. Fast sorting implements a second layer of coincidence requirements between specific detector channels to clean up background. There are two default fast sorting windows, one for the ion chamber and one for SABRE. The ion chamber requires that the scintillator and anode be within a certain time window, while the SABRE requires that any sabre channel and the scintillator be within a certain window.

Finally, the sorted data can be analyzed by the SFPAnalyzer class to generate some basic physics data. Note that analyzing will typically bloat the size of the file due to the large number of parameters that are generated. The analyzing step is required to use the plotter tool.

3.3 Plotting

Plotting is done using the SFPPlotter class and the CutHandler class. The plotter generates an un-cut and a cut set of histograms. The SFPPlotter is where users will most likely make the most changes, as the desired plots can change from experiement to experiment. Plots can be added using the `MyFill` function, which wraps the creation, storage, and filling of ROOT histograms. Simply add a `MyFill` where ever you would like to make a histogram. Cuts are given using the CutList file format. Cuts are ROOT `TCutG` objects stored in ROOT files; these `TCutG` should have the default name (CUTG). The CutList can handle any number of cuts, where the cut should have a name, file, and x and y variables specified in the list. X and Y variable names need to be defined in the CutHandler class in the InitVariableMap function. By default the variables x1, x2, xavg, scintLeft, cathode, and anodeBack are all defined.

3.4 Scalers

The event builder supports defining certain digitizer channels to be scalers (similar to nscldaq). As is currently setup, the program counts the hits in the scaler file and then saves that number to a TParameter in the final output file of the event builder. Scalers must be defined in a the ScalerFile, using the format described in the examples given.

3.5 GUI

The event builder has a GUI which can simplify use for overnight runs. However, it should be noted that any changes to the code have the potential to break GUI functionality. If you want to make changes, and want to not build the GUI, simply modify the make file by removing the EVBEXE from the all command. Now only the command line version (GWMEVB_CL) will be made.

3.6 ROOT Dictionary

In order to store more complicated data than simple doubles or ints in ROOT trees, a dictionary must be generated. The dictionary requires two header files, one of which is called `DataStructs.h`, which holds the actual `C++` definitions, and the other of which is `LinkDef_sps.h`, which holds the precompiler instructions for ROOT. When the program is built using the normal make commands, the dictionary is automatically generated and compiled. The ROOT dictionary must then be loaded into each subsequent analysis that uses the data file from the event builder. For ease of use, a dynamic library of the dictionary is also automatically generated. To use the dynamic library in a ROOT macro, simply add the line `R__LOAD_LIBRARY(<path_to_evb>/lib/libSPSDict.so)` after the include statements. For use in any compiled `C++` simply link it as a normal library.

4 Building and Running

To build the code use the following command from the event builder directory:

```
make
```

To run the event builder gui then use the command:

```
./bin/GWMEVB
```

To run the command line version use:

```
./bin/GWMEVB_CL <operation> <your_input_file.txt>
```

If the code needs rebuilt use:

```
make clean
```

If you want it completely wiped use:

```
make clean_header && make clean
```

5 Example data

For testing purposes, example data is given (as well as an example workspace) in the example directory. The data is from a SABRE-SPS run on $^{12}C(^{3}He,\alpha)^{11}C$ at 20 degrees with 24 MeV beam energy and 9511.0 G B-field. You can use this data to perform some basic tests; there are no expected decay products from ^{11}C at these settings, so there should be no real SABRE coincidences. Channel and shift maps are included in the etc directory for this data
(`ChannelMap_March2020_newFormat_092020.txt`,
`ShiftMap_April2020_newFormat_10102020.txt`)

6 Common Problems and Questions

- **What do I do when the event builder crashes?**
  When the event builder crashes there are three things to look at; the first is the ROOT error report in the terminal, the second is the program report in the terminal, and the last is the `temp_binary` directory. The first will tell you whether or not the event builder crashed due to the ROOT gui environment. Typically you have to scroll through a bunch of error code and then you'll see a line reporting Bad Window. This is not the fault of the event builder code, but rather the ROOT packages. If this is not the case, then keep scrolling to see where in the pipeline the program crashed. Most cases of bad behavior have some sort of report to the terminal. Finally, when the evb crashes, it typically does not finish cleaning up (deleting) the files in `\temp_binary`. Make sure you manually delete them all before re-running the event builder.

- **Why does xavg look awful?**
  Make sure you've put the values into the event builder for the kinematic correction with the right units.

- **The event builder is taking a really long time, how can I speed it up?**
  Event building large amounts of data is a time consuming process, but there are some precautions that can be taken. Shorter runs, while more work for shifters, can be built more quickly, as the event builder scales exponentially with data size. As it is currently constructed, the event builder only constructs one type of event and scalers. If you have data that *should not* be coincident with other parts of the experiment, this will slow down the building process, as it doesn't follow the same rules as other pieces. A good example of this is a monitor detector in the scattering chamber. To overcome this, make that channel a scaler, or don't include it in the archive, and analyze it separately.

- **All of the histograms are empty, what do I do?**
  First is to make sure that there is actually data in the analyzed file. Open this and check the basic histograms it made. If that file is empty, this typically points to a channel map issue, so make sure that you've defined all channels properly. If that doesn't work, try tweaking the coincidence window. If all shifts have been applied correctly, the window can typically run as small as 1.5 μs, but wider (2.0 μs) is a bit safer when the shifts aren't as dialed in. If none of that works, run the builder without the analysis, and see if those files have any data in them. This will help tell you if there is an issue at the beginning or the end of the pipeline.

- **I made a change to the code and now it won't compile/run, but everything seems correct. What do I do?**
  First, try a `make clean` and `make` cycle. Changes, especially to header

files, can cause some disagreements between modules in the code. If this fails, take another look at your changes and make sure they're compatible with existing code.